A Theoretician's Guide to the Experimental Analysis of Algorithms

David S. Johnson
AT&T Research

April 30, 1996

1 Introduction

There has been much encouragement recently for the experimental analysis of algorithms. This encouragement has come both from those like myself who are already actively experimenting and from funding agencies who view experimentation as providing a pathway from theory into practice. Unfortunately, many who attempt to enter the field of experimental analysis discover that it is fraught with pitfalls. In many ways, the implementation of an algorithm is the easy part. The hard part is successfully using that implementation to produce meaningful and valuable (and publishable!) research results. This note will attempt to provide some suggestions and guiding principles that may help would-be experimenters to be successful.

For the purposes of argument, there are perhaps three basic reasons why one it might be worthwhile to implement an algorithm, each giving rise to a different type of paper (although many papers end up being hybrids of more than one type).

1. To use the code in a particular application. This typically gives rise to an *application paper*, whose purpose is to describe the usefulness of the algorithm in that context.

2. To make the algorithm available for others to use. This often gives rise to what one might call a *salespitch paper*, where results are published for standard benchmark instances to illustrate the desirability of this algorithm in comparison to previous competitors.

3. To learn about how the algorithm behaves in practices/on average. This is the motivation behind what I will call an *experimental paper*.

This note will concentrate on the experimental paper, although many of the principles I discuss will be applicable to the other types as well The experimental paper is the type that is closest in spirit to the typical goals of a paper in the theoretical analysis of algorithms. It is also the type I have the most experience with, both as an author and as a reader, referee, and editor. I have unfortunately seen far more defective papers than good ones, and even fairly good ones typically receive referee's reports suggesting significant revisions, often involving substantial additional experiments. In this note I shall attempt to draw on this experience by articulating a set of general principles applicable to experimental papers and explaining how and how not to satisfy them

In the next section I state these principles and elaborate upon them. Section 3 then provides a list of some of the more egregious defects that have occurred in experimental papers, pointing out which principles they violate. Most of this paper will concentrate on the study of algorithms for problems in the standard format where both instances and outputs are finite objects, and the key metrics are resource usage (typically time and memory) and (in the case of approximation algorithms) quality of the output solution. Many key computational problems do not fit this model, and in Section 4 I describe some of these and consider what the appropriate principles should be in such domains.

Several of the principles enunciated in this paper may be somewhat controversial, at least as I elaborate them. Indeed, although there is much common agreement on what makes good experimental analysis of algorithms, certain aspects have been the subject of much debate, such as the relevance of running time comparisons. For other points of view, see such recent papers as [?] [?] [?] [?] [?] [?].

2 Principles

2.1 Perform Newsworthy Experiments

This principle presumably applies to all scientific papers: the results should be new and of interest/use to some reasonably-sized collection of readers. However, the standards for "interest" are more stringent for experimental algorithms papers.

For instance, it is much harder to justify experimental work on problems with no direct applications than it is to justify theoretical work on such problems. In theoretical work, one often chooses to work on problems because of their amenability to analysis (and in the hope that they are representative enough that the lessons learned or proof techniques developed may be more widely applicable). In contrast, most problems are amenable to experimental analysis of some sort, so more selectivity is required. Moreover, problems without applications do not have real-world instances, so the experimenter is left to invent (and justify) test data in a vacuum. And referees may well question the testing of code for algorithms that will never be used in practice.

A second dimension of "newsworthiness" has to do with the algorithms studied. A key question one would like answered is whether a given algorithm might be a useful practical alternative. Since theoretical analysis (especially worst-case analysis) often cannot be relied on for an answer, one can justify studying a wide variety of algorithms, at least to the point where that question is answered (and assuming there was at least some plausible reason why one might have hoped for practicality). Of course when the answer is a definitive *No*, as for example with a *dominated* approximation algorithm that is slower and produces worse solutions than some already well-studied competitor, there will be little value in further study once this point is established. Note, however, that domination is not always easy to establish, given how dependent running times can be on implementations or machine/operating system interactions, and in many cases simply getting within a factor of two of another algorithm may be enough to claim competitiveness. Furthermore, *practical value* has many aspects, and one could well justify studying a dominated algorithm if for example it were much easier to implement, or if codes were already available and in widespread use. In any case, however, it is up to you to make a convincing case for the potential usefulness of any algorithm you choose to study in detail.

A third dimension of newsworthiness has to do with the generality and relevance (and credibility) of the results obtained and the conclusions derivable from them. One is of necessity limited to a finite number of tests, but for a paper to be interesting, the results should yield broader insights, either in terms of asymptotics or presumptions of robustness, as we shall elaborate on under Principle 2.4 below. (This is one way in which an experimental paper aspires to higher goals than a salespitch paper, which can settle for the simple conclusion that the tested algorithm produces the stated results on the set of benchmark instances.) Also, the conclusion *that* algorithm A outperforms algorithm B becomes more interesting (and more believable) when supported by measurements (operation counts, subroutine profiling, etc.) that show in detail *why* this happens.

2.2 Tie Your Paper to the Literature

One key component in establishing the newsworthiness of a paper is placing it in proper context with respect to the literature on the problem being studied. Indeed, before undertaking any experimental project, one should do one's best to discover and thoroughly study the prior literature if it exists. This should go without saying, but surprisingly many papers have failed because of lack of diligence on this point. Moreover, knowing what has done can not only save you from performing "uninteresting" experiments, it can also suggest what the interesting questions should be. What behavior needs explaining? What algorithms seem open to further improvement? What types of test instances have not yet been adequately studied?

When your problem does have a literature, there are obligations as well as opportunities, however. One must of course provide comparisons between one's results and those obtained previously, and here is another case where the experimenter's job may be more difficult than the theoretician's. The potential difficulty is that direct comparisons may be much more problematic. Although earlier papers may have dealt with the same algorithm and similar test instances, they may not have dealt with precisely the same test instances or with the same implementations of the algorithm, and they most likely did not provide running times on the same machine/operating system combination.

Ideally, if one's goal is to provide comparisons to (or more insights into the behavior of) a previously-studied algorithm, one should obtain the code for that algorithm that was used in the previous experiments, and report results for that code on one's own machine (while verifying that those results are consistent with those reported for the code previously). If that is not possible, the next best option is to develop a comparable implementation of the previous algorithm which one can run on one's own machine and which, based on comparisons with the previously-reported results, seems to be of comparable effectiveness. In the unfortunate event that the two implementations do not appear to be of comparable effectiveness, for example when one appears to run in quadratic time and the other in linear time, this point should be noted and explanations provided if possible (and your implementation should be the linear time one; see the next principle).

The least desirable option, but one to which one must occasionally resort when attempting to provide comparisons to a hard-to-implement or incompletely-specified algorithm from the literature, is simply to compare one's results on one's own machine and test instances to those reported in the previous paper. In some cases, where the previous experiments were performed on some unfamiliar architecture, this may be impossible. Typically, however, one can provide significantly accurate ballpark bounds on relative machine speed to at least answer the question of whether running times are possible competitive or whether one implementation is clearly faster than the other. Even such rough comparisons are better than nothing. Readers need to see how your paper fits into the literature, even if the fit is at best only approximate.

2.3 Use Efficient Implementations

This is a perhaps controversial principle, but has much to recommend it. The point is that the efficiency of the code you test should be (at least to within a constant factor in practice) as good as that of a code you would expect people to use in the real world. This is important for several reasons. First, efficient implementations make it easier for you to support claims of practicality. Second, results for implementations significantly different from those one would use in practice may yield distorted pictures. Third, faster implementations allow one to perform experiments on more and/or larger instances, or to finish the study more quickly.

Note that this principle does not imply one need go overboard on efficiency issues and fine tune one's code mercilessly to save the last 10Nor does it mean one must implement all the theoretical refinements that yield small improvement in worst-case running time, especially if there's no indication that such tweaking will have a major practical impact. One should, however, make efficient use of data structures when this is called for and when real-world implementers would be likely to do so, e.g., $\Theta(N^2)$ sorting algorithms should typically be avoided.

And one can make exceptions. If, for instance, you are examining a well-studied algorithm whose newsworthiness has already been established, your study is not primarily interested in running time, and cycles are less-expensive for you than programming time, then you can conceivably get by with a simple but inefficient implementation (at least until the referees start asking for more data...)

2.4 Use Testbeds that Can Support General Conclusions

There are basically two types of test instances available to experimenters: particular instances from real-world applications (or pseudo-applications) and randomly generated instances. The former are typically found in standard repositories, such as `TSPLIB`, but might come from private sources proprietary to the author. The latter are provided by instance generators that, given a seed and a list of instance parameters, such as size and perhaps other parameters relating to instance structure, produce a random instance consistent with those parameters. Note that random instances need not be structureless (like random graphs), and indeed should preferably be structured in ways that reflect some aspects of real-world instances.

A key property of random instance generators is that they can generate instances of arbitrarily large size. This enables one to get a partial handle on the types of asymptotic questions typically addressed by theoretical analysis, although machine dependent effects such as cache limitations will typically make precise asymptotics impossible. What one *can* determine, however, is the biggest instance you can reasonably run on your machine, and some idea about how much larger the runnable instances could be on a next-generation machine with greater speed, memory, etc., which can be a key practical question.

This gives random instances an advantage in experimental studies, although one that can be substantially dissipated if there is no evidence that the random instance class says anything about what will happen for real-world instances. A perhaps ideal combination is a study of one (or more) random instance classes, combined with results for real-world instances that enable the reader to evaluate the predictive quality of the random results. Especially useful are parameterized random instance generators which allow you to investigate the effect of structure as well as size on performance. (In a few well-established domains, the study of simple but unrealistic distributions has gained credibility simply as a question of mathematical interest. In these cases there can be some justification for experimentation simply as a way of generating mathematical conjectures, although this perhaps should be viewed more as an example of experimental mathematics than experimental analysis of algorithms.)

Papers that study only real-world instances have a harder task in establishing the generality of their applicability, although if the set of real-world instances is broad and varied enough, one can draw conclusions about the robustness of various performance measures across instance types.

2.5 Provide Explanations and Measurements to Back them Up

In the real-world, the key performance characteristics of an algorithm are it's resource requirements (running time, memory usage, etc.) and (with approximation algorithms) the quality of the solutions it generates. It is thus natural to concentrate on measurements of these quantities when performing a first experimental analysis. However, if our goal is understanding as it should be in an experimental paper, we must soon turn to the question of explaining why the observed measurements occur. (Plausible explanations also add to the credibility of the experimental results themselves.)

Explanations are at their most believable if backed up by compelling evidence. When a worst-case $\Theta(N^2)$ algorithm appears to be taking $\Theta(N \log N)$ in practice, what part of the worst-case analysis proved to be overly pessimistic in practice? Why does this seemingly more complicated algorithm take less time than this supposedly simple and efficient one. Often these questions can be answered by performing additional measurements, such as operation counts or profiling of time spent in subroutine calls. (Measurements of this sort have also on more than one occasion led to the "explanation" that an implementation was defective, something one would definitely like to know before publishing results about it.)

2.6 Ensure Reproducibility

As with all scientific studies, a key part of an experimental paper is the reproducibility of the results. One of such a paper's job is to describe algorithms, test instances, computing environment, results, etc. in enough detail so that a reader could at least in principle perform similar experiments that would lead to the same basic conclusions. With the growing popularity and accessibility of the Internet, it is now possible to offload part of this responsibility to a Web site, thus allowing readers direct access to codes, instances generators, test instances, and even raw data. However, as of now it is still important that the paper provide enough information on its own so as to enable rough replication. Indeed, your conclusions will be better supported if researchers can reproduce them using slightly different instances and and their own implementations.

This principle also imposes requirements on the study reported by the paper, since unless the study covered a wide enough range of instances (and a large enough number of runs in the case of randomized algorithms), the conclusions drawn may well be wrong, and hence irreproducible for a different reason. The amount of data needed will vary of course with the detail of the conclusions. A first paper on a given algorithm may well settle for trying to indicate general trends in its performance without quantifying them, and this may require fewer test instances than a subsequent paper that wishes for instance to determine constants of proportionality in its asymptotic running time.

2.7 Ensure Comparability and Give the Full Picture

Following this principle will make it possible for subsequent researchers to compare their results to yours, just as I've asked you to compare your work to that done in the past. It may also enable future researchers and practitioners to learn things from your results that you didn't even recognize yourself.

First, although it is typically overkill to present all your raw data, you should provide meaningful summaries. If one is to present averages, one should report over how many instance (and how many runs in the case of randomized algorithms) and information about the distribution. The latter might simply be standard deviations if the data appears roughly normally distributed, but might need to be more pictorial (histograms, boxplots) if the distribution is more idiosyncratic. And any scaling or normalizing of measurements should be carefully explained so that the raw averages can be regenerated if desired. In particular, although pictorial summaries of data can often be the best way to demonstration one's overall conclusions, this should not typically be the only presentation of the data. Tables of data values may be relegated to an appendix, but should be included.

Second, one should not hide anomalous results, either by omitting them or by failing to point them out when they are present. Ideally, one should provide explanations for them, but even if you have none, the reader should know of their existence. Even if you can't see it, they may be the key to major insights into algorithms, implementations, or test data.

Finally, even if your main subject is not running times, you should report yours. As noted above, running time is an important component of newsworthiness, and so a reader may legitimately want to know this information before committing to a detailed study of your results. Similarly, even if your main subject is only one aspect of the running time (say the local optimization phase), you need to provide information on all aspects of the running time (including preprocessing time that is shared across multiple runs on the same instance) so that readers can get a picture of what is involved in running the full algorithm in practice.

3 Common Mistakes

3.1 Studying totally-dominated algorithms in detail

3.2 Failing to tell the reader about key implementation details

3.3 Extrapolating from tiny testbeds

3.4 Using irreproducible benchmarks

3.5 Using running time as a stopping criterion

3.6 Ignoring hidden costs (such as preprocessing)

3.7 Using poor programming ability as an excuse

3.8 Failing to describe key details of implementations, machines, etc.

3.9 Misusing statistical tools

3.10 Failing to use graphical methods when appropriate

3.11 Obscuring raw data by presenting it in hard-to-read charts

3.12 Comparing apples and oranges

3.13 Drawing conclusions the data doesn't support

3.14 Leaving obvious anomalies unnoted and unexplained

3.15 Failing to support explanations with measurements/supplementary experiments

3.16 Ignoring the literature (the self-referential study)

4 Extensions to Other Types of Problem Domain

- How do you evaluate metaheuristic approaches?
- How do you evaluate program checkers/correctors etc.?
- How do you evaluate system dynamic algorithms with a conceptually infinite time horizon?
- How do you evaluate algorithms where other metrics are relevant than time, space, and solution quality (such as rate of convergence, etc.)?

5 Referneces

[1] W. S. Barr, B. L. Golden, J. P. Kelly, M. G. C. Resende, and W. R. Stewart, "Designing and reporting on computational experiments with heuristic methods," *J. Heuristics* **1:1** (1995), 9-32.

[2] C. C. McGeoch, "Toward an experimental method for algorithm simulation," *INFORMS J. Computing* **8:1** (1996), 1-15.